```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```

import pandas as pd
csv_path = './content/MyDrive/drive/data/csv'
artist_train = pd.read_csv('/content/drive/MyDrive/data/csv/Artist/artist_train')
# lets visualize one imag
base_url = '/content/drive/MyDrive/data/images'
# lets start creating data
artist = '/content/drive/MyDrive/data/csv/Artist'
genre = '/content/drive/MyDrive/data/csv/Genre'
style = '/content/drive/MyDrive/data/csv/Style'
data_dir = '/content/drive/MyDrive/data/csv'

artist_train_path = data_dir + '/artist_train.csv'
artist_val_path = data_dir + '/artist_val.csv'
artist_class_path = data_dir + '/artist_class.txt'

genre_train_path = data_dir + '/genre_train.csv'
genre_val_path = data_dir + '/genre_val.csv'
genre_class_path = data_dir + '/genre_class.txt'

style_train_path = data_dir + '/style_train.csv'
style_val_path = data_dir + '/style_val.csv'
style_class_path = data_dir + '/style_class.txt'

artist_train = pd.read_csv(data_dir + '/artist_train.csv')
artist_val = pd.read_csv(data_dir + '/artist_val.csv')
artist_class = pd.read_csv(artist_class_path, header=None, names=["artist_name"])

genre_train = pd.read_csv(data_dir + '/genre_train.csv')
genre_val = pd.read_csv(data_dir + '/genre_val.csv')
genre_class = pd.read_csv(genre_class_path, header=None, names=["genre_name"])


style_train = pd.read_csv(data_dir + '/style_train.csv')
style_val = pd.read_csv(data_dir + '/style_val.csv')
style_class = pd.read_csv(style_class_path, header=None, names=["style_name"])

# genre_class['genre_name'][1]
len(style_class)
```

```
27
```

```
import os
import pandas as pd
import torch
from torch.utils.data import Dataset
from torchvision import transforms
from PIL import Image
from collections import defaultdict
import random
from tqdm import tqdm
import matplotlib.pyplot as plt

# Define dataset class
class BalancedArtDataset(Dataset):
    def __init__(self, csv_file, img_dir, class_mapping, transform=None, images_per_class=32):
        self.data = pd.read_csv(csv_file)
        self.img_dir = img_dir
        self.class_mapping = class_mapping
        self.transform = transform
        self.images_per_class = images_per_class

        # Filter out missing images
        print("Filtering missing images...")
        self.data = self.data[self.data.iloc[:, 0].apply(lambda x: os.path.exists(os.path.join(img_dir, str(x))))]

        # Group images by class
        print("Grouping images by class...")
        self.class_images = defaultdict(list)
```

```
        for _, row in tqdm(self.data.iterrows(), total=len(self.data), desc="Processing rows"):
            self.class_images[row.iloc[1]].append(row)

        # Balance dataset with 32 images per class
        print("Balancing dataset...")
        self.final_data = []
        all_images = []
        for cls, images in tqdm(self.class_images.items(), total=len(self.class_images), desc="Processing classes"):
            if len(images) >= images_per_class:
                selected_images = random.sample(images, images_per_class)
            else:
                selected_images = images[:]
                all_images.extend(images)  # Store extra images for filling
            self.final_data.extend(selected_images)

        # Fill missing slots with extra images
        print("Filling missing slots...")
        needed_images = images_per_class * len(self.class_images) - len(self.final_data)
        if needed_images > 0:
            self.final_data.extend(random.sample(all_images, min(needed_images, len(all_images))))

        # Shuffle dataset
        print("Shuffling dataset...")
        random.shuffle(self.final_data)

        # Count images per class
        self.class_counts = defaultdict(int)
        for row in self.final_data:
            self.class_counts[row.iloc[1]] += 1

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

    def __getitem__(self, idx):
        row = self.final_data[idx]
        img_path = os.path.join(self.img_dir, str(row.iloc[0]))
        label = row.iloc[1]
        image = Image.open(img_path).convert("RGB")

        if self.transform:
            image = self.transform(image)

        return image, label


    def visualize_class_distribution(self):
        plt.figure(figsize=(12, 6))
        plt.bar(self.class_counts.keys(), self.class_counts.values(), color='skyblue')
        plt.xlabel("Class")
        plt.ylabel("Number of Images")
        plt.title("Class Distribution in Balanced Dataset")
        plt.xticks(rotation=45)
        plt.show()

    def visualize_samples(self, num_samples=10):
        fig, axes = plt.subplots(1, num_samples, figsize=(40, 20))
        for i in range(num_samples):
            image, label = self.__getitem__(random.randint(0, len(self) - 1))
            image = image.permute(1, 2, 0).numpy()  # Convert to (H, W, C)
            image = (image * 0.5) + 0.5  # Unnormalize
            axes[i].imshow(image)
            axes[i].set_title(f"Class: {label} {genre_class['genre_name'][label]}")
            axes[i].axis("off")
        plt.show()

    # Function to compare artist and genre relationships



# Define transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])

# Create balanced artist dataset
```

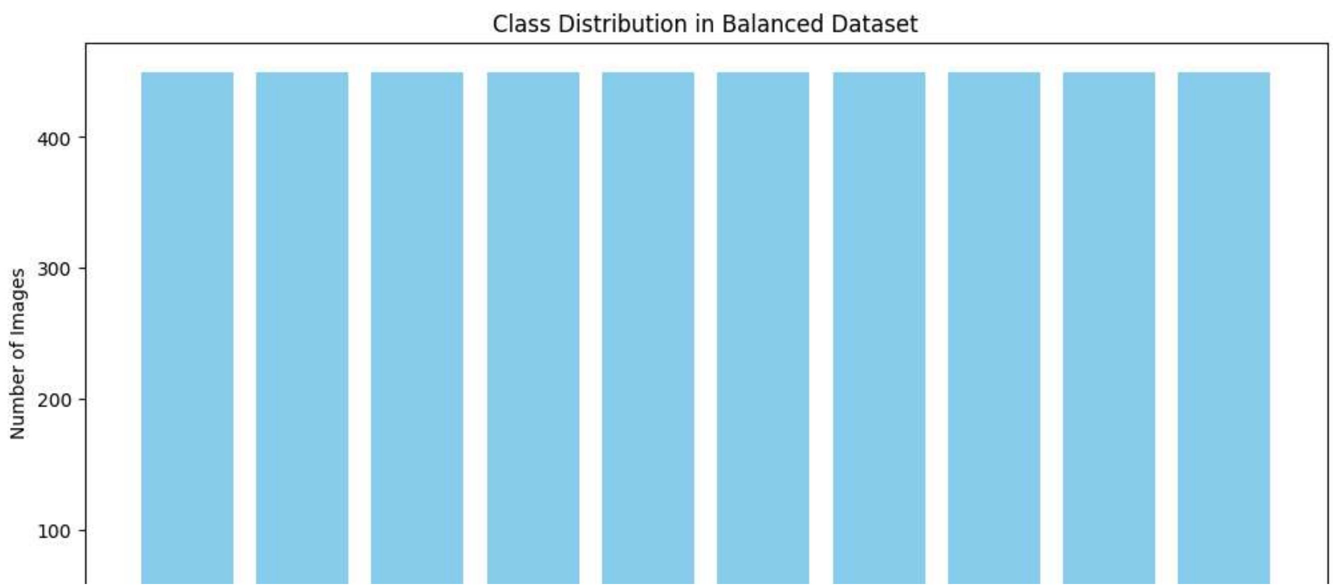
```
print("Creating balanced artist dataset...")
genre_balanced_dataset = BalancedArtDataset(genre_train_path, "/content/drive/MyDrive/data/images", genre_class_path, transform=transfc
genre_test_balanced_dataset = BalancedArtDataset(genre_val_path, "/content/drive/MyDrive/data/images", genre_class_path, transform=trar

# Check dataset length
print("Artist balanced dataset size:", len(genre_balanced_dataset), len(genre_test_balanced_dataset))

# Visualize class distribution
genre_balanced_dataset.visualize_class_distribution()

# Visualize sample images
genre_balanced_dataset.visualize_samples(num_samples=10)
```

```
Creating balanced artist dataset...
Filtering missing images...
Grouping images by class...
Processing rows: 100%|██████████| 45501/45501 [00:03<00:00, 13441.81it/s]
Balancing dataset...
Processing classes: 100%|██████████| 10/10 [00:00<00:00, 2948.75it/s]
Filling missing slots...
Shuffling dataset...
Filtering missing images...
Grouping images by class...
Processing rows: 100%|██████████| 19492/19492 [00:01<00:00, 19431.19it/s]
Balancing dataset...
Processing classes: 100%|██████████| 10/10 [00:00<00:00, 5752.71it/s]
Filling missing slots...
Shuffling dataset...
Artist balanced dataset size: 4500 1500
```




```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.models as models
from torch.utils.data import DataLoader
from tqdm import tqdm

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

# Define DataLoaders
batch_size = 32  # Adjust based on GPU memory

train_loader = DataLoader(genre_balanced_dataset, batch_size=batch_size, shuffle=True, num_workers=4,pin_memory=True)
val_loader = DataLoader(genre_test_balanced_dataset, batch_size=batch_size, shuffle=False, num_workers=4, pin_memory=True)

print("Dataloaders created successfully!")
```

```
Using device: cuda
Dataloaders created successfully!
/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:624: UserWarning: This DataLoader will create 4 worker processes
  warnings.warn(
```

```python
# Import necessary libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim.lr_scheduler import ReduceLROnPlateau
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
from tqdm import tqdm
import time

# Hyperparameters
num_epochs_fc = 5        # Train FC layer only for 5 epochs
num_epochs_finetune = 6  # Fine-tune entire model for 6 more epochs
initial_lr = 0.001       # Learning rate for training FC layer
finetune_lr = 1e-4       # Lower learning rate for fine-tuning
batch_size = 32
weight_decay = 1e-4      # L2 regularization

# Define the number of output classes
num_classes = len(genre_class)  # Replace with your actual class count

# Load the ResNet-50 model pre-trained on ImageNet
model = models.resnet50(weights=models.ResNet50_Weights.DEFAULT)
model.fc = nn.Linear(model.fc.in_features, num_classes).to(device)  # Replace the FC layer to match your task

model = model.to(device)  # Move model to the appropriate device (CPU/GPU)

# Define loss function and optimizer for training the FC layer
criterion = nn.CrossEntropyLoss().to(device)
fc_optimizer = optim.Adam(model.fc.parameters(), lr=initial_lr, weight_decay=weight_decay)

# Function to train the fully connected (FC) layer only
def train_fc_layer():
    print("Starting Step 1: Training FC Layer Only...")

    # Freeze all layers except the FC layer
    for param in model.parameters():
        param.requires_grad = False

    # Ensure the new FC layer is trainable
    for param in model.fc.parameters():
        param.requires_grad = True

    model.train()  # Set the model to training mode

    for epoch in range(num_epochs_fc):
        running_loss, correct, total = 0.0, 0, 0
        loop = tqdm(train_loader, leave=True, desc=f"Epoch [{epoch+1}/{num_epochs_fc}]")

        for images, labels in loop:
            images, labels = images.to(device), labels.to(device)

            # Zero the gradients
            fc_optimizer.zero_grad()

            # Forward pass
            outputs = model(images)

            # Compute loss
            loss = criterion(outputs, labels)
            loss.backward()  # Backpropagation
            fc_optimizer.step()  # Update FC layer weights

            running_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

            # Update the progress bar
            loop.set_postfix(loss=running_loss / len(train_loader), acc=100 * correct / total)

        print(f"Epoch [{epoch+1}/{num_epochs_fc}], Loss: {running_loss / len
```

```
            (train_loader):.4f},
              f"Accuracy: {100 * correct / total:.2f}%")

    print("🎯 Step 1 Complete: FC Layer Training Finished!")


# Function to fine-tune the entire ResNet-50 network
def fine_tune():
    print("Starting Step 2: Fine-Tuning the Entire Network...")

    # Unfreeze all layers
    for param in model.parameters():
        param.requires_grad = True

    # Define optimizer and learning rate scheduler for fine-tuning
    finetune_optimizer = optim.Adam(model.parameters(), lr=finetune_lr, weight_decay=weight_decay)
    scheduler = ReduceLROnPlateau(finetune_optimizer, mode='min', factor=0.1, patience=3, verbose=True)

    best_val_loss = float('inf')

    for epoch in range(num_epochs_finetune):
        model.train()
        running_loss, correct, total = 0.0, 0, 0
        loop = tqdm(train_loader, leave=True, desc=f"Epoch [{epoch+1}/{num_epochs_finetune}]")

        for images, labels in loop:
            images, labels = images.to(device), labels.to(device)

            finetune_optimizer.zero_grad()

            # Forward pass
            outputs = model(images)

            # Compute loss
            loss = criterion(outputs, labels)
            loss.backward()
            finetune_optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

            # Update the progress bar
            loop.set_postfix(loss=running_loss / len(train_loader), acc=100 * correct / total)

        # Validation phase
        val_loss, val_acc = evaluate(val_loader)
        scheduler.step(val_loss)

        print(f"Epoch [{epoch+1}/{num_epochs_finetune}], "
              f"Train Loss: {running_loss / len(train_loader):.4f}, Train Acc: {100 * correct / total:.2f}%, "
              f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%")

        # Save the best model based on validation loss
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict(), "/content/drive/MyDrive/art_model/best_genre_resnet50_model.pth")
            print("✅ Best model saved!")

    print("🎯 Step 2 Complete: Fine-Tuning Finished!")


# Function to evaluate the model on validation data
def evaluate(loader):
    model.eval()
    val_loss, correct, total = 0.0, 0, 0

    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)

```

```
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    avg_loss = val_loss / len(loader)
    accuracy = 100 * correct / total
    return avg_loss, accuracy


# Start training
train_fc_layer()  # Step 1: Train FC Layer only
fine_tune()       # Step 2: Fine-tune the entire ResNet-50 model

print("🎉 Training complete! The best model is saved as 'best_resnet50_model.pth'.")

```

```
Starting Step 1: Training FC Layer Only...
Epoch [1/5]: 100%|██████████| 141/141 [07:49<00:00,  3.33s/it, acc=55.4, loss=1.45]
Epoch [1/5], Loss: 1.4543, Accuracy: 55.36%
Epoch [2/5]: 100%|██████████| 141/141 [01:16<00:00,  1.83it/s, acc=68.2, loss=1.01]
Epoch [2/5], Loss: 1.0138, Accuracy: 68.24%
Epoch [3/5]: 100%|██████████| 141/141 [01:18<00:00,  1.81it/s, acc=71.8, loss=0.905]
Epoch [3/5], Loss: 0.9046, Accuracy: 71.82%
Epoch [4/5]: 100%|██████████| 141/141 [01:17<00:00,  1.83it/s, acc=74, loss=0.819]
Epoch [4/5], Loss: 0.8189, Accuracy: 74.02%
Epoch [5/5]: 100%|██████████| 141/141 [01:19<00:00,  1.78it/s, acc=75.6, loss=0.775]
/usr/local/lib/python3.11/dist-packages/torch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is deprecated. Please use get
  warnings.warn(
Epoch [5/5], Loss: 0.7755, Accuracy: 75.58%
🎯 Step 1 Complete: FC Layer Training Finished!
Starting Step 2: Fine-Tuning the Entire Network...
Epoch [1/6]: 100%|██████████| 141/141 [01:27<00:00,  1.61it/s, acc=76.9, loss=0.678]
Epoch [1/6], Train Loss: 0.6778, Train Acc: 76.89%, Val Loss: 0.8499, Val Acc: 72.67%
✅ Best model saved!
Epoch [2/6]: 100%|██████████| 141/141 [01:28<00:00,  1.58it/s, acc=95.8, loss=0.181]
Epoch [2/6], Train Loss: 0.1805, Train Acc: 95.78%, Val Loss: 0.9826, Val Acc: 70.47%
Epoch [3/6]: 100%|██████████| 141/141 [01:27<00:00,  1.62it/s, acc=99.4, loss=0.0472]
Epoch [3/6], Train Loss: 0.0472, Train Acc: 99.44%, Val Loss: 1.0296, Val Acc: 71.73%
Epoch [4/6]: 100%|██████████| 141/141 [01:25<00:00,  1.65it/s, acc=99.6, loss=0.0241]
Epoch [4/6], Train Loss: 0.0241, Train Acc: 99.64%, Val Loss: 1.1336, Val Acc: 72.13%
Epoch [5/6]: 100%|██████████| 141/141 [01:27<00:00,  1.61it/s, acc=100, loss=0.0134]
Epoch [5/6], Train Loss: 0.0134, Train Acc: 99.96%, Val Loss: 1.1553, Val Acc: 72.67%
Epoch [6/6]: 100%|██████████| 141/141 [01:27<00:00,  1.62it/s, acc=100, loss=0.00832]
Epoch [6/6], Train Loss: 0.0083, Train Acc: 99.96%, Val Loss: 1.1345, Val Acc: 72.67%
🎯 Step 2 Complete: Fine-Tuning Finished!
🎉 Training complete! The best model is saved as 'best_resnet50_model.pth'.
```