Mount Google Drive

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Import Libraries and Define Paths

```
import os, torch, json, shutil, numpy as np, pandas as pd
from glob import glob
from PIL import Image
from torch.utils.data import random_split, Dataset, DataLoader
from torchvision import transforms as T
torch.manual_seed(2024)

# Define file paths
train_csv_path = "/content/drive/MyDrive/Dataset/train/other/train.csv"
label_map_path = "/content/drive/MyDrive/Dataset/train/other/label_num_to_disease_map.json"
train_images_path = "/content/drive/MyDrive/Dataset/train/train_data"
model_save_path = "/content/drive/MyDrive/Dataset/train/crop_disease_models"

# Create model save directory if it doesn't exist
os.makedirs(model_save_path, exist_ok=True)
```

Define CustomDataset Class

```
class CustomDataset(Dataset):
    def __init__(self, root, transformations=None):
        self.transformations = transformations
        # Read meta data
        meta_data = pd.read_csv(train_csv_path)
        # Read json file and get class names dictionary
        with open(label_map_path) as json_file:
            data = json.load(json_file)
        self.cls_names = {int(key): value for key, value in data.items()}
        # Get image file names and their corresponding labels
        im_names = list(meta_data["image_id"])
        gt_names = list(meta_data["label"])
        self.meta_data = {}
        self.cls_counts, count, data_count = {}, 0, 0
        for idx, im_path in enumerate(im_names):
            fname = f"{train_images_path}/{im_path}"
            if not os.path.isfile(fname):
                continue
            else:
                class_name = self.cls_names[int(gt_names[idx])]
                if class_name not in self.cls_counts:
                    self.cls_counts[class_name] = 1
                    count += 1
                else:
                    self.cls_counts[class_name] += 1
                self.meta_data[fname] = int(gt_names[idx])

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

    def __getitem__(self, idx):
        im_path = list(self.meta_data.keys())[idx]
        im = Image.open(im_path).convert("RGB")
        gt = self.meta_data[im_path]
        if self.transformations is not None:
            im = self.transformations(im)
        return im, gt
```

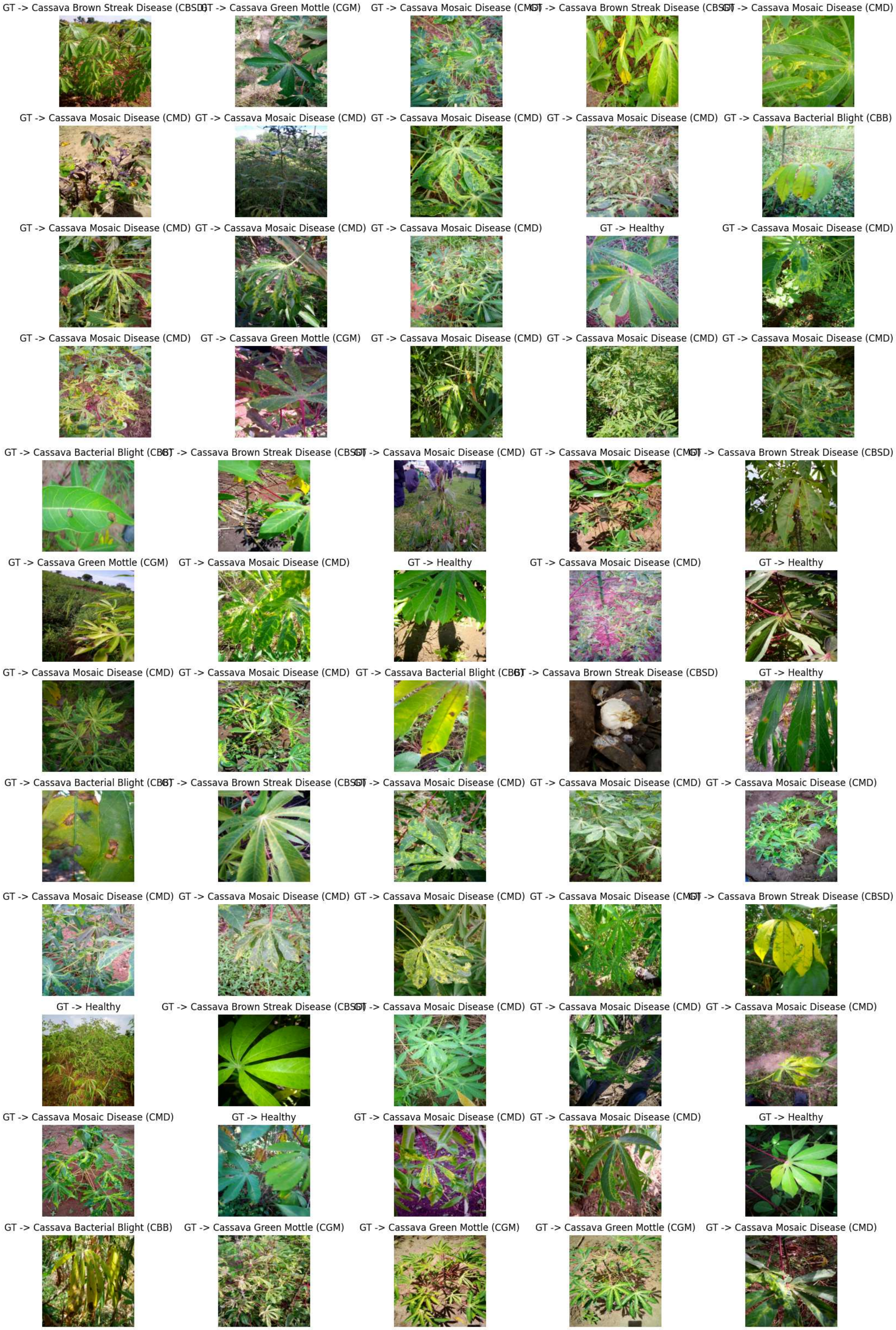
Define get_dls Function and Load Data

```
def get_dls(root, transformations, bs, split=[0.9, 0.05, 0.05], ns=4):
    ds = CustomDataset(root=root, transformations=transformations)
    total_len = len(ds)
    tr_len = int(total_len * split[0])
    vl_len = int(total_len * split[1])
    ts_len = total_len - (tr_len + vl_len)
    tr_ds, vl_ds, ts_ds = random_split(dataset=ds, lengths=[tr_len, vl_len, ts_len])
    tr_dl, val_dl, ts_dl = (
        DataLoader(tr_ds, batch_size=bs, shuffle=True, num_workers=ns),
        DataLoader(vl_ds, batch_size=bs, shuffle=False, num_workers=ns),
        DataLoader(ts_ds, batch_size=1, shuffle=False, num_workers=ns),
    )
    return tr_dl, val_dl, ts_dl, ds.cls_names


mean, std, im_size = [0.485, 0.456, 0.406], [0.229, 0.224, 0.225], 224
tfs = T.Compose([T.Resize((im_size, im_size)), T.ToTensor(), T.Normalize(mean=mean, std=std)])

tr_dl, val_dl, ts_dl, classes = get_dls(
    root=train_images_path, transformations=tfs, bs=32
)

print(len(tr_dl))
print(len(val_dl))
print(len(ts_dl))
for key, value in classes.items():
    print(f"{key}: {value}")
```

```
476
27
846
0: Cassava Bacterial Blight (CBB)
1: Cassava Brown Streak Disease (CBSD)
2: Cassava Green Mottle (CGM)
3: Cassava Mosaic Disease (CMD)
4: Healthy
/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:617: UserWarning: This DataLoader will create 4 worker proces
  warnings.warn(
```

Data Visualization Functions

```
import random
from matplotlib import pyplot as plt

def tensor_2_im(t, t_type="rgb"):
    gray_tfs = T.Compose(
        [
            T.Normalize(mean=[0.0], std=[1 / 0.5]),
            T.Normalize(mean=[-0.5], std=[1]),
        ]
    )
    rgb_tfs = T.Compose(
        [
            T.Normalize(mean=[0.0, 0.0, 0.0], std=[1 / 0.229, 1 / 0.224, 1 / 0.225]),
            T.Normalize(mean=[-0.485, -0.456, -0.406], std=[1.0, 1.0, 1.0]),
        ]
    )
    invTrans = gray_tfs if t_type == "gray" else rgb_tfs
    return (
        (invTrans(t) * 255)
        .detach()
        .squeeze()
        .cpu()
        .permute(1, 2, 0)
        .numpy()
        .astype(np.uint8)
        if t_type == "gray"
        else (invTrans(t) * 255).detach().cpu().permute(1, 2, 0).numpy().astype(np.uint8)
    )


def visualize(data, n_ims, rows, cmap=None, cls_names=None):
    assert cmap in ["rgb", "gray"], "Rasmni oq-qora yoki rangli ekanini aniqlashtirib bering!"
    if cmap == "rgb":
        cmap = "viridis"
    plt.figure(figsize=(20, 10))
    indekslar = [random.randint(0, len(data) - 1) for _ in range(n_ims)]
    for idx, indeks in enumerate(indekslar):
        im, gt = data[indeks]
        # Start plot
        plt.subplot(rows, n_ims // rows, idx + 1)
        if cmap:
            plt.imshow(tensor_2_im(im, cmap), cmap=cmap)
        else:
            plt.imshow(tensor_2_im(im))
        plt.axis("off")
        if cls_names is not None:
            plt.title(f"GT -> {cls_names[int(gt)]}")
        else:
            plt.title(f"GT -> {gt}")


visualize(tr_dl.dataset, 20, 4, "rgb", list(classes.values()))
visualize(val_dl.dataset, 20, 4, "rgb", list(classes.values()))
visualize(ts_dl.dataset, 20, 4, "rgb", list(classes.values()))
```

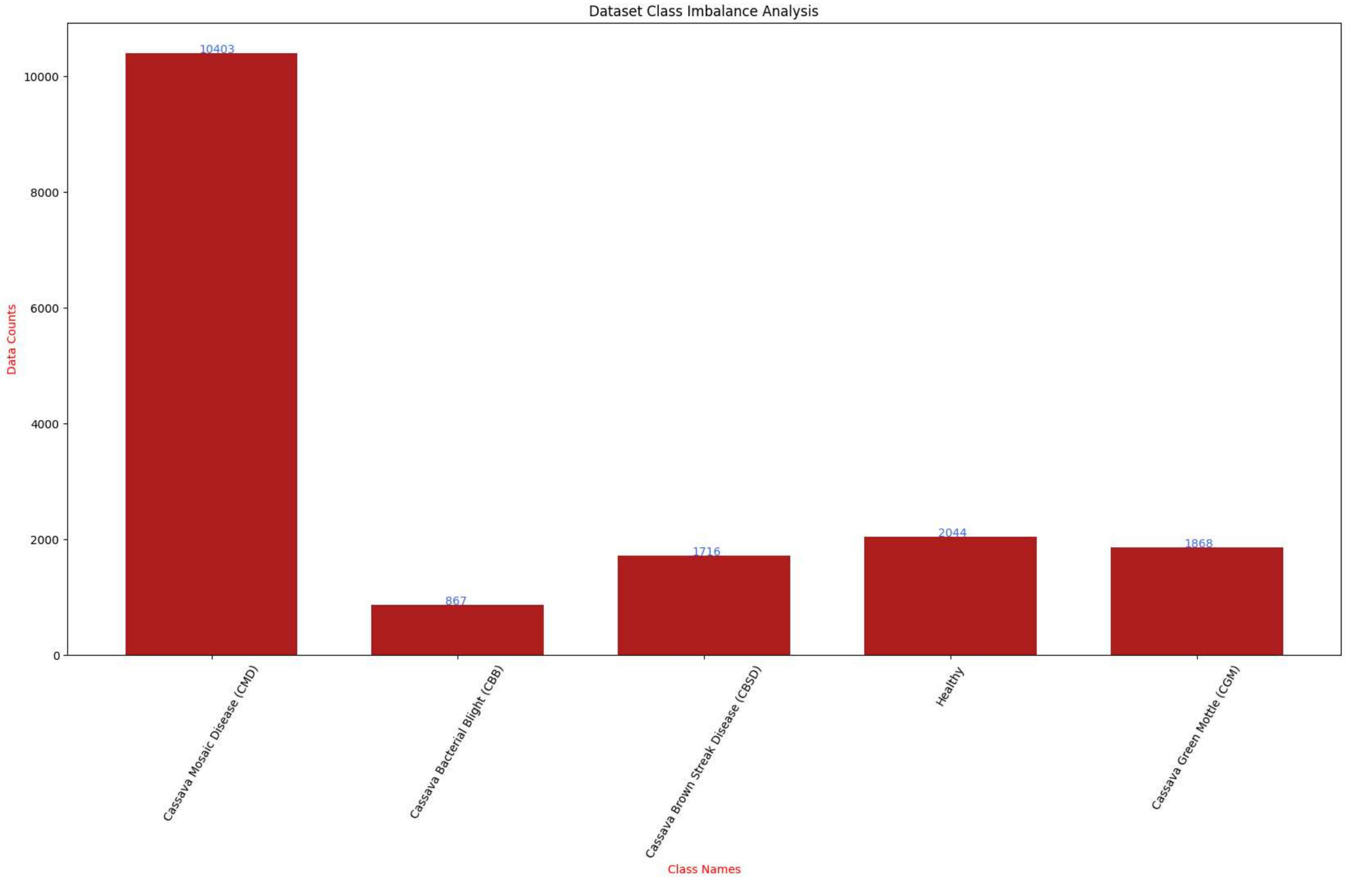

GT -> Cassava Brown Streak Disease (CBSD)
GT -> Cassava Green Mottle (CGM)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Brown Streak Disease (CBSD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Bacterial Blight (CBB)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Healthy
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Green Mottle (CGM)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Bacterial Blight (CBB)
GT -> Cassava Brown Streak Disease (CBSD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Brown Streak Disease (CBSD)
GT -> Cassava Green Mottle (CGM)
GT -> Cassava Mosaic Disease (CMD)
GT -> Healthy
GT -> Cassava Mosaic Disease (CMD)
GT -> Healthy
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Bacterial Blight (CBB)
GT -> Cassava Brown Streak Disease (CBSD)
GT -> Healthy
GT -> Cassava Bacterial Blight (CBB)
GT -> Cassava Brown Streak Disease (CBSD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Brown Streak Disease (CBSD)
GT -> Healthy
GT -> Cassava Brown Streak Disease (CBSD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Healthy
GT -> Cassava Mosaic Disease (CMD)
GT -> Cassava Mosaic Disease (CMD)
GT -> Healthy
GT -> Cassava Bacterial Blight (CBB)
GT -> Cassava Green Mottle (CGM)
GT -> Cassava Green Mottle (CGM)
GT -> Cassava Green Mottle (CGM)
GT -> Cassava Mosaic Disease (CMD)

Data Analysis Function

```
def data_analysis(root, transformations):
    ds = CustomDataset(root=root, transformations=transformations)
    cls_counts, width, text_width = ds.cls_counts, 0.7, 0.05
    text_height = 2
    cls_names = list(cls_counts.keys())
    counts = list(cls_counts.values())
    fig, ax = plt.subplots(figsize=(20, 10))
    indices = np.arange(len(counts))
    ax.bar(indices, counts, width, color="firebrick")
    ax.set_xlabel("Class Names", color="red")
    ax.set_xticklabels(cls_names, rotation=60)
    ax.set(xticks=indices, xticklabels=cls_names)
    ax.set_ylabel("Data Counts", color="red")
    ax.set_title(f"Dataset Class Imbalance Analysis")
    for i, v in enumerate(counts):
        ax.text(i - text_width, v + text_height, str(v), color="royalblue")
```

```
data_analysis(root=train_images_path, transformations=tfs)
```

```
<ipython-input-6-b5ebb4d7a923>:11: UserWarning: set_ticklabels() should only be used with a fixed number of ticks, i.e. after set_ti
  ax.set_xticklabels(cls_names, rotation=60)
```



Model Training and Validation

```
!pip install torchmetrics
import timm, torchmetrics
from tqdm import tqdm

# Create the model
m = timm.create_model("rexnet_150", pretrained=True, num_classes=len(classes))

# Initialize device
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")
def train_setup(m, device):
    return (
        m.to(device).eval(),  # Move model to the specified device
        10,                   # Number of epochs
        device,               # Device ("cuda" or "cpu")
        torch.nn.CrossEntropyLoss(),
        torch.optim.Adam(params=m.parameters(), lr=3e-4),
    )
def to_device(batch, device):
    return batch[0].to(device), batch[1].to(device)

def get_metrics(model, ims, gts, loss_fn, epoch_loss, epoch_acc, epoch_f1, f1_score):
    preds = model(ims)
    loss = loss_fn(preds, gts)
    return (
        loss
```

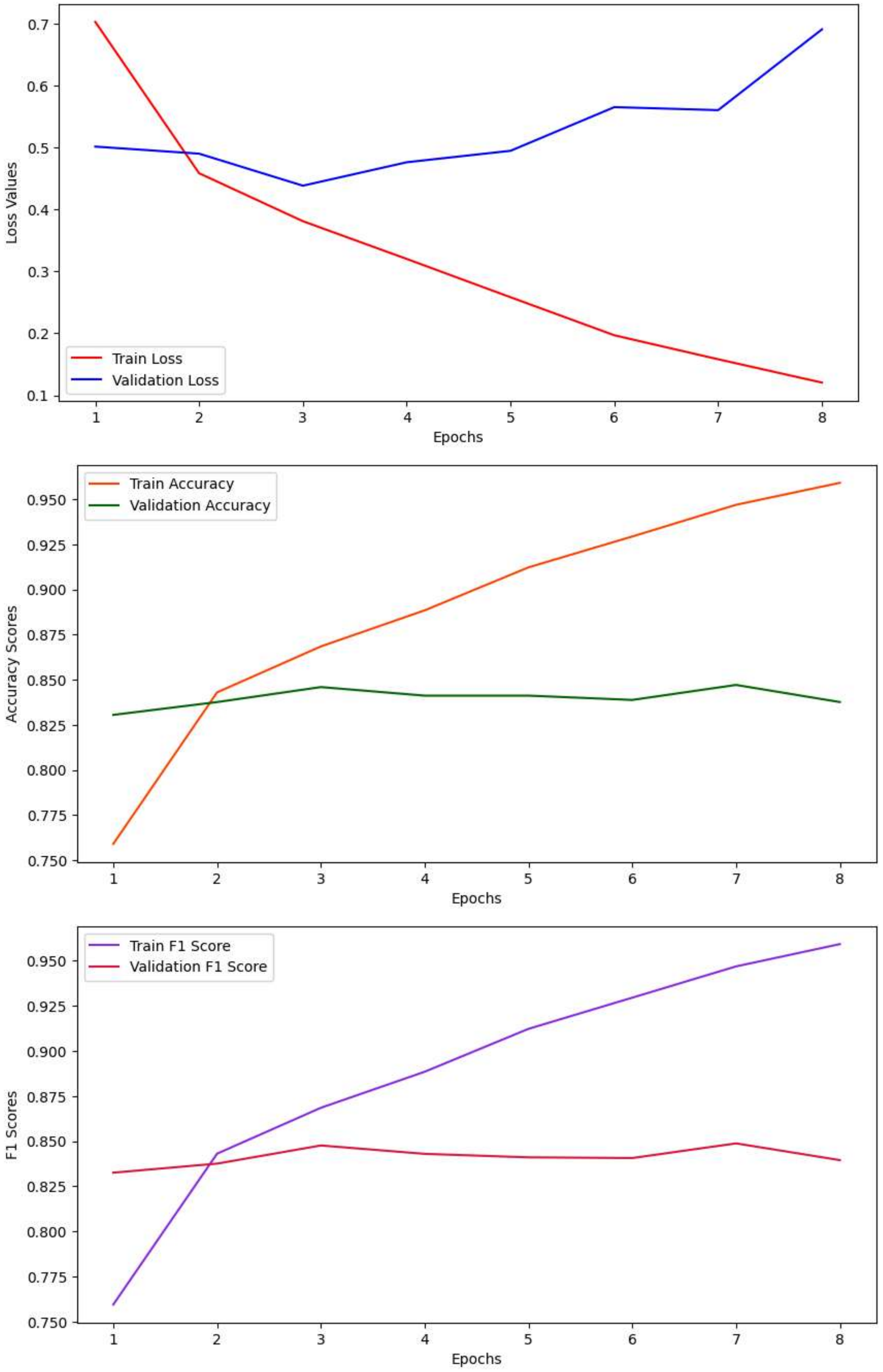
```
            loss,
            epoch_loss + (loss.item()),
            epoch_acc + (torch.argmax(preds, dim=1) == gts).sum().item(),
            epoch_f1 + f1_score(preds, gts),
        )

    # Initialize training parameters
    m, epochs, device, loss_fn, optimizer = train_setup(m, device)
    f1_score = torchmetrics.F1Score(task="multiclass", num_classes=len(classes)).to(device)
    save_prefix, save_dir = "crop", model_save_path
    os.makedirs(save_dir, exist_ok=True)

    print("\n" + "=" * 50)
    print(f"{'Starting Training':^50}")
    print("=" * 50 + "\n")

    best_acc, best_loss, threshold, not_improved, patience = 0, float("inf"), 0.01, 0, 5
    tr_losses, val_losses, tr_accs, val_accs, tr_f1s, val_f1s = [], [], [], [], [], []
    best_loss = float("inf")

    for epoch in range(epochs):
        epoch_loss, epoch_acc, epoch_f1 = 0, 0, 0

        # Training loop
        total_images = len(tr_dl.dataset)  # Total number of training images
        processed_images = 0  # Counter for processed images
        with tqdm(enumerate(tr_dl), total=len(tr_dl), desc=f"Epoch {epoch + 1} - Training") as pbar:
            for idx, batch in pbar:
                ims, gts = to_device(batch, device)
                loss, epoch_loss, epoch_acc, epoch_f1 = get_metrics(
                    m, ims, gts, loss_fn, epoch_loss, epoch_acc, epoch_f1, f1_score
                )
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                # Update processed image count
                processed_images += ims.size(0)

                # Update progress bar with total and processed image counts
                pbar.set_postfix({
                    "Total Images": total_images,
                    "Processed": processed_images,
                    "Loss": f"{loss.item():.3f}"
                })

        # Track training metrics
        tr_loss_to_track = epoch_loss / len(tr_dl)
        tr_acc_to_track = epoch_acc / len(tr_dl.dataset)
        tr_f1_to_track = epoch_f1 / len(tr_dl)
        tr_losses.append(tr_loss_to_track)
        tr_accs.append(tr_acc_to_track)
        tr_f1s.append(tr_f1_to_track)

        print("\n" + "-" * 50)
        print(f"{'Training Results':^50}")
        print("-" * 50)
        print(f"Epoch {epoch + 1} - Loss: \033[91m{tr_loss_to_track:.3f}\033[0m")
        print(f"Epoch {epoch + 1} - Accuracy: \033[92m{tr_acc_to_track:.3f}\033[0m")
        print(f"Epoch {epoch + 1} - F1 Score: \033[94m{tr_f1_to_track:.3f}\033[0m")
        print("-" * 50 + "\n")

        # Validation loop
        m.eval()
        total_val_images = len(val_dl.dataset)  # Total number of validation images
        processed_val_images = 0  # Counter for processed validation images
        with torch.no_grad():
            val_epoch_loss, val_epoch_acc, val_epoch_f1 = 0, 0, 0
            for idx, batch in enumerate(val_dl):
                ims, gts = to_device(batch, device)
                loss, val_epoch_loss, val_epoch_acc, val_epoch_f1 = get_metrics(
                    m, ims, gts, loss_fn, val_epoch_loss, val_epoch_acc, val_epoch_f1, f1_score
                )

                # Update processed validation image count
                processed_val_images += ims.size(0)

            # Track validation metrics
            val_loss_to_track = val_epoch_loss / len(val_dl)
            val_acc_to_track = val_epoch_acc / len(val_dl.dataset)
            val_f1_to_track = val_epoch_f1 / len(val_dl)
            val_losses.append(val_loss_to_track)
            val_accs.append(val_acc_to_track)
            val_f1s.append(val_f1_to_track)
        val_f1s.append(val_f1_to_track)

        print("\n" + "-" * 50)
        print(f"{'Validation Results':^50}")
        print("-" * 50)
        print(f"Epoch {epoch + 1} - Loss: \033[91m{val_loss_to_track:.3f}\033[0m")
        print(f"Epoch {epoch + 1} - Accuracy: \033[92m{val_acc_to_track:.3f}\033[0m")
        print(f"Epoch {epoch + 1} - F1 Score: \033[94m{val_f1_to_track:.3f}\033[0m")
        print(f"Total Validation Images: {total_val_images}")
        print(f"Processed Validation Images: {processed_val_images}")
        print("-" * 50 + "\n")

        # Save the best model
        if val_loss_to_track < (best_loss + threshold):
            best_loss = val_loss_to_track
            torch.save(m.state_dict(), f"{save_dir}/{save_prefix}_best_model.pth")
            print(f"\033[92mModel saved at epoch {epoch + 1} with loss {best_loss:.3f}\033[0m\n")
        else:
            not_improved += 1
            print(f"\033[93mLoss did not decrease for {not_improved} epochs\033[0m")
            if not_improved == patience:
                print(f"\033[91mStopping training due to no improvement for {patience} epochs.\033[0m")
                break

    print("\n" + "=" * 50)
    print(f"{'Training Completed':^50}")
    print("=" * 50 + "\n")
```

```
Epoch 6 - Accuracy: 0.839
Epoch 6 - F1 Score: 0.841
Total Validation Images: 844
Processed Validation Images: 844
--------------------------------------------------

Loss did not decrease for 3 epochs
Epoch 7 - Training: 100%|██████████| 476/476 [03:15<00:00,  2.43it/s, Total Images=15208, Processed=15208, Loss=0.346]

--------------------------------------------------
                 Training Results
--------------------------------------------------
Epoch 7 - Loss: 0.158
Epoch 7 - Accuracy: 0.947
Epoch 7 - F1 Score: 0.947
--------------------------------------------------



--------------------------------------------------
                Validation Results
--------------------------------------------------
Epoch 7 - Loss: 0.560
Epoch 7 - Accuracy: 0.847
Epoch 7 - F1 Score: 0.849
Total Validation Images: 844
Processed Validation Images: 844
--------------------------------------------------

Loss did not decrease for 4 epochs
Epoch 8 - Training: 100%|██████████| 476/476 [03:15<00:00,  2.43it/s, Total Images=15208, Processed=15208, Loss=0.009]
--------------------------------------------------
                 Training Results
--------------------------------------------------
Epoch 8 - Loss: 0.121
Epoch 8 - Accuracy: 0.959
Epoch 8 - F1 Score: 0.959
--------------------------------------------------



--------------------------------------------------
                Validation Results
--------------------------------------------------
Epoch 8 - Loss: 0.691
Epoch 8 - Accuracy: 0.838
Epoch 8 - F1 Score: 0.840
Total Validation Images: 844
Processed Validation Images: 844
--------------------------------------------------

Loss did not decrease for 5 epochs
Stopping training due to no improvement for 5 epochs.

==================================================
                Training Completed
==================================================
```

Learning Curves Visualization

```
class PlotLearningCurves:
    def __init__(self, tr_losses, val_losses, tr_accs, val_accs, tr_f1s, val_f1s):
        self.tr_losses, self.val_losses, self.tr_accs, self.val_accs, self.tr_f1s, self.val_f1s = (
            tr_losses,
            val_losses,
            tr_accs,
            val_accs,
            tr_f1s,
            val_f1s,
        )

    def plot(self, array_1, array_2, label_1, label_2, color_1, color_2):
        plt.plot(array_1, label=label_1, c=color_1)
        plt.plot(array_2, label=label_2, c=color_2)

    def create_figure(self):
        plt.figure(figsize=(10, 5))

    def decorate(self, ylabel, xlabel="Epochs"):
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.xticks(
            ticks=np.arange(len(self.tr_accs)),
            labels=[i for i in range(1, len(self.tr_accs) + 1)],
        )
        plt.legend()
        plt.show()

    def visualize(self):
        # Figure 1
        self.create_figure()
        self.plot(
            array_1=self.tr_losses,
            array_2=self.val_losses,
            label_1="Train Loss",
            label_2="Validation Loss",
            color_1="red",
            color_2="blue",
        )
        self.decorate(ylabel="Loss Values")

        # Figure 2
        self.create_figure()
        self.plot(
            array_1=self.tr_accs,
            array_2=self.val_accs,
            label_1="Train Accuracy",
            label_2="Validation Accuracy",
            color_1="orangered",
            color_2="darkgreen",
        )
        self.decorate(ylabel="Accuracy Scores")

        # Figure 3
        self.create_figure()
        self.plot(
            array_1=[tr_f1.cpu() for tr_f1 in self.tr_f1s],
            array_2=[vl_f1.cpu() for vl_f1 in self.val_f1s],
            label_1="Train F1 Score",
            label_2="Validation F1 Score",
            color_1="blueviolet",
            color_2="crimson",
        )
        self.decorate(ylabel="F1 Scores")
```

```
PlotLearningCurves(tr_losses, val_losses, tr_accs, val_accs, tr_f1s, val_f1s).visualize()
```



Inference and GradCAM Visualization

```
import cv2


class SaveFeatures:
    """Extract pretrained activations"""
```