```
# IMPORTANT: SOME KAGGLE DATA SOURCES ARE PRIVATE
# RUN THIS CELL IN ORDER TO IMPORT YOUR KAGGLE DATA SOURCES.
import kagglehub
kagglehub.login()
```

```
# IMPORTANT: RUN THIS CELL IN ORDER TO IMPORT YOUR KAGGLE DATA SOURCES,
# THEN FEEL FREE TO DELETE THIS CELL.
# NOTE: THIS NOTEBOOK ENVIRONMENT DIFFERS FROM KAGGLE'S PYTHON
# ENVIRONMENT SO THERE MAY BE MISSING LIBRARIES USED BY YOUR
# NOTEBOOK.

feyzazkefe_trashnet_path = kagglehub.dataset_download('feyzazkefe/trashnet')
haseeburrehmanmarwat_testimageglass_path = kagglehub.dataset_download('haseeburrehmanmarwat/testimageglass')
haseeburrehmanmarwat_glassimage_path = kagglehub.dataset_download('haseeburrehmanmarwat/glassimage')
haseeburrehmanmarwat_trashimage_path = kagglehub.dataset_download('haseeburrehmanmarwat/trashimage')
haseeburrehmanmarwat_videodata_path = kagglehub.dataset_download('haseeburrehmanmarwat/videodata')

print('Data source import complete.')
```

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import copy
import random
```

```
# Device configuration
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Paths
data_dir = '/kaggle/input/trashnet/dataset-resized'
```

```
# Data augmentation (stronger)
transform_train = transforms.Compose([
    transforms.Resize((160, 160)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(20),
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.1),
    transforms.RandomAffine(10, translate=(0.05, 0.05)),
    transforms.RandomPerspective(distortion_scale=0.2, p=0.5),
    transforms.RandomResizedCrop(128, scale=(0.75, 1.0)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

transform_test = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

```
# Load dataset
full_dataset = datasets.ImageFolder(data_dir, transform=transform_train)
class_names = full_dataset.classes
```

```
# Split dataset into train, val, test
train_size = int(0.7 * len(full_dataset))
val_size = int(0.15 * len(full_dataset))
test_size = len(full_dataset) - train_size - val_size
train_data, val_data, test_data = torch.utils.data.random_split(full_dataset, [train_size, val_size, test_size])

val_data.dataset.transform = transform_test
test_data.dataset.transform = transform_test

train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
val_loader = DataLoader(val_data, batch_size=32, shuffle=False)
test_loader = DataLoader(test_data, batch_size=32, shuffle=False)
```

```
# Improved deeper CNN architecture
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
        )
        self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.classifier = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(256, len(class_names))
        )

    def forward(self, x):
        x = self.features(x)
        x = self.global_avg_pool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x

model = CNN().to(device)
```

```
# Loss, optimizer, scheduler
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0005, weight_decay=1e-4)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', patience=4, factor=0.5, verbose=True)
```

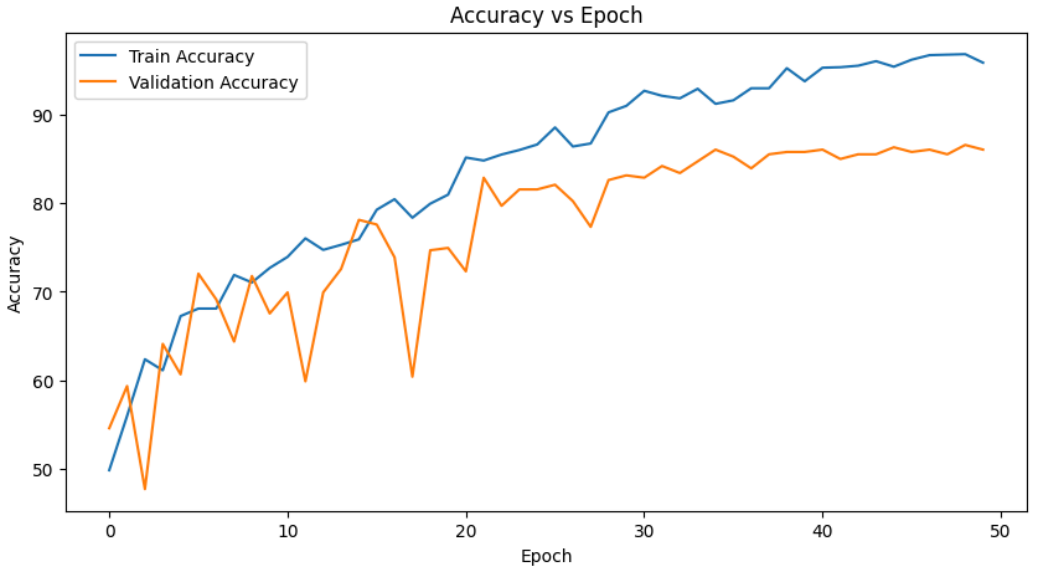
```
# Train the model
num_epochs = 50
train_losses, val_losses, train_accs, val_accs = [], [], [], []
best_val_acc = 0
best_model_wts = copy.deepcopy(model.state_dict())

for epoch in range(num_epochs):
    model.train()
    running_loss, correct, total = 0.0, 0, 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

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

    train_losses.append(running_loss / len(train_loader))
    train_accs.append(100 * correct / total)

    model.eval()
    val_loss, correct, total = 0.0, 0, 0
```

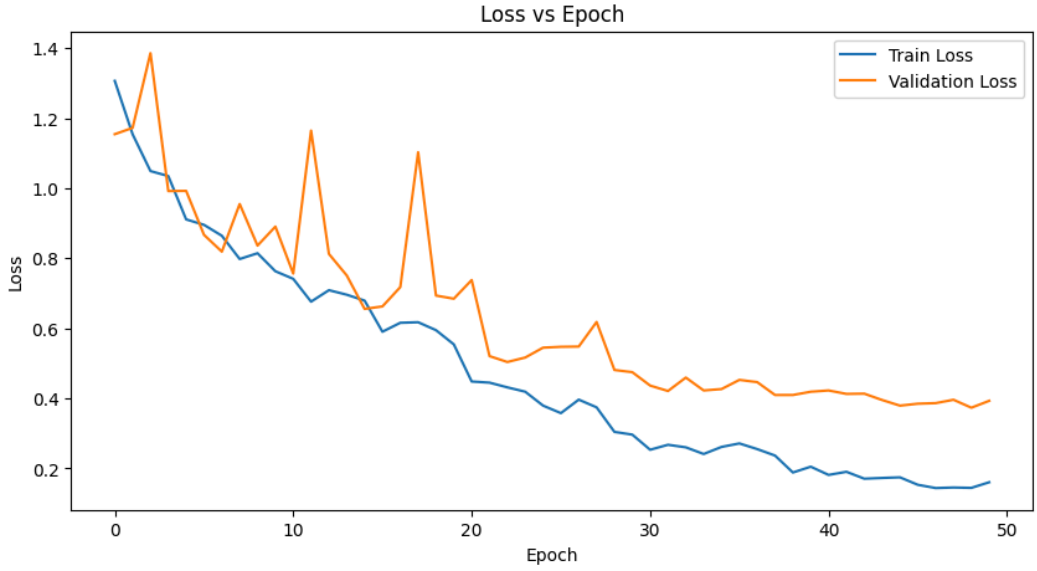
```
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    val_losses.append(val_loss / len(val_loader))
    val_acc = 100 * correct / total
    val_accs.append(val_acc)

    scheduler.step(val_loss / len(val_loader))

    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_model_wts = copy.deepcopy(model.state_dict())

    print(f"Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_losses[-1]:.4f}, Val Loss: {val_losses[-1]:.4f}, Train Acc: {train_accs[-1]:.2f}%, Val A
```

```
Epoch [1/50], Train Loss: 1.3069, Val Loss: 1.1546, Train Acc: 49.89%, Val Acc: 54.62%
Epoch [2/50], Train Loss: 1.1544, Val Loss: 1.1727, Train Acc: 56.00%, Val Acc: 59.37%
Epoch [3/50], Train Loss: 1.0493, Val Loss: 1.3860, Train Acc: 62.39%, Val Acc: 47.76%
Epoch [4/50], Train Loss: 1.0350, Val Loss: 0.9922, Train Acc: 61.14%, Val Acc: 64.12%
Epoch [5/50], Train Loss: 0.9112, Val Loss: 0.9927, Train Acc: 67.25%, Val Acc: 60.69%
Epoch [6/50], Train Loss: 0.8957, Val Loss: 0.8672, Train Acc: 68.10%, Val Acc: 72.03%
Epoch [7/50], Train Loss: 0.8646, Val Loss: 0.8191, Train Acc: 68.10%, Val Acc: 69.13%
Epoch [8/50], Train Loss: 0.7981, Val Loss: 0.9552, Train Acc: 71.89%, Val Acc: 64.38%
Epoch [9/50], Train Loss: 0.8146, Val Loss: 0.8364, Train Acc: 71.04%, Val Acc: 71.77%
Epoch [10/50], Train Loss: 0.7632, Val Loss: 0.8907, Train Acc: 72.68%, Val Acc: 67.55%
Epoch [11/50], Train Loss: 0.7415, Val Loss: 0.7566, Train Acc: 73.93%, Val Acc: 69.92%
Epoch [12/50], Train Loss: 0.6763, Val Loss: 1.1647, Train Acc: 76.02%, Val Acc: 59.89%
Epoch [13/50], Train Loss: 0.7091, Val Loss: 0.8127, Train Acc: 74.72%, Val Acc: 69.92%
Epoch [14/50], Train Loss: 0.6962, Val Loss: 0.7511, Train Acc: 75.28%, Val Acc: 72.56%
Epoch [15/50], Train Loss: 0.6793, Val Loss: 0.6556, Train Acc: 75.90%, Val Acc: 78.10%
Epoch [16/50], Train Loss: 0.5904, Val Loss: 0.6627, Train Acc: 79.24%, Val Acc: 77.57%
Epoch [17/50], Train Loss: 0.6160, Val Loss: 0.7178, Train Acc: 80.43%, Val Acc: 73.88%
Epoch [18/50], Train Loss: 0.6175, Val Loss: 1.1030, Train Acc: 78.34%, Val Acc: 60.42%
Epoch [19/50], Train Loss: 0.5949, Val Loss: 0.6937, Train Acc: 79.92%, Val Acc: 74.67%
Epoch [20/50], Train Loss: 0.5542, Val Loss: 0.6847, Train Acc: 80.94%, Val Acc: 74.93%
Epoch [21/50], Train Loss: 0.4483, Val Loss: 0.7381, Train Acc: 85.12%, Val Acc: 72.30%
Epoch [22/50], Train Loss: 0.4450, Val Loss: 0.5208, Train Acc: 84.79%, Val Acc: 82.85%
Epoch [23/50], Train Loss: 0.4316, Val Loss: 0.5040, Train Acc: 85.46%, Val Acc: 79.68%
Epoch [24/50], Train Loss: 0.4192, Val Loss: 0.5167, Train Acc: 85.97%, Val Acc: 81.53%
Epoch [25/50], Train Loss: 0.3795, Val Loss: 0.5449, Train Acc: 86.60%, Val Acc: 81.53%
Epoch [26/50], Train Loss: 0.3580, Val Loss: 0.5474, Train Acc: 88.52%, Val Acc: 82.06%
Epoch [27/50], Train Loss: 0.3965, Val Loss: 0.5481, Train Acc: 86.37%, Val Acc: 80.21%
Epoch [28/50], Train Loss: 0.3745, Val Loss: 0.6183, Train Acc: 86.71%, Val Acc: 77.31%
Epoch [29/50], Train Loss: 0.3044, Val Loss: 0.4812, Train Acc: 90.21%, Val Acc: 82.59%
Epoch [30/50], Train Loss: 0.2966, Val Loss: 0.4750, Train Acc: 90.95%, Val Acc: 83.11%
Epoch [31/50], Train Loss: 0.2534, Val Loss: 0.4367, Train Acc: 92.65%, Val Acc: 82.85%
Epoch [32/50], Train Loss: 0.2676, Val Loss: 0.4212, Train Acc: 92.08%, Val Acc: 84.17%
Epoch [33/50], Train Loss: 0.2603, Val Loss: 0.4596, Train Acc: 91.80%, Val Acc: 83.38%
Epoch [34/50], Train Loss: 0.2413, Val Loss: 0.4227, Train Acc: 92.87%, Val Acc: 84.70%
Epoch [35/50], Train Loss: 0.2614, Val Loss: 0.4267, Train Acc: 91.18%, Val Acc: 86.02%
Epoch [36/50], Train Loss: 0.2715, Val Loss: 0.4529, Train Acc: 91.57%, Val Acc: 85.22%
Epoch [37/50], Train Loss: 0.2553, Val Loss: 0.4465, Train Acc: 92.93%, Val Acc: 83.91%
Epoch [38/50], Train Loss: 0.2367, Val Loss: 0.4097, Train Acc: 92.93%, Val Acc: 85.49%
Epoch [39/50], Train Loss: 0.1884, Val Loss: 0.4098, Train Acc: 95.19%, Val Acc: 85.75%
Epoch [40/50], Train Loss: 0.2050, Val Loss: 0.4191, Train Acc: 93.72%, Val Acc: 85.75%
Epoch [41/50], Train Loss: 0.1816, Val Loss: 0.4227, Train Acc: 95.25%, Val Acc: 86.02%
Epoch [42/50], Train Loss: 0.1905, Val Loss: 0.4129, Train Acc: 95.31%, Val Acc: 84.96%
Epoch [43/50], Train Loss: 0.1707, Val Loss: 0.4136, Train Acc: 95.48%, Val Acc: 85.49%
Epoch [44/50], Train Loss: 0.1729, Val Loss: 0.3954, Train Acc: 95.98%, Val Acc: 85.49%
Epoch [45/50], Train Loss: 0.1746, Val Loss: 0.3794, Train Acc: 95.36%, Val Acc: 86.28%
Epoch [46/50], Train Loss: 0.1533, Val Loss: 0.3849, Train Acc: 96.15%, Val Acc: 85.75%
Epoch [47/50], Train Loss: 0.1440, Val Loss: 0.3865, Train Acc: 96.66%, Val Acc: 86.02%
Epoch [48/50], Train Loss: 0.1456, Val Loss: 0.3962, Train Acc: 96.72%, Val Acc: 85.49%
Epoch [49/50], Train Loss: 0.1445, Val Loss: 0.3734, Train Acc: 96.78%, Val Acc: 86.54%
Epoch [50/50], Train Loss: 0.1606, Val Loss: 0.3932, Train Acc: 95.81%, Val Acc: 86.02%
```

```
# Load best model weights
model.load_state_dict(best_model_wts)

# Plot accuracy
plt.figure(figsize=(10,5))
plt.plot(train_accs, label='Train Accuracy')
plt.plot(val_accs, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy vs Epoch')
plt.show()
```

```
# Plot loss
plt.figure(figsize=(10,5))
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss vs Epoch')
plt.show()
```






```
# Testing the model
model.eval()
correct, total = 0, 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Test Accuracy: {100 * correct / total:.2f}%')
```

```
Test Accuracy: 86.84%
```

```
# Show 10 test samples with predictions
samples = list(test_data)
random.shuffle(samples)
samples = samples[:5]
```

```
plt.figure(figsize=(20, 10))
for i, (image, label) in enumerate(samples):
    img_tensor = image.unsqueeze(0).to(device)
    output = model(img_tensor)
    _, pred = torch.max(output.data, 1)
    pred_label = class_names[pred.item()]
    actual_label = class_names[label]

    image = image.permute(1, 2, 0).cpu().numpy()
    image = (image * 0.5) + 0.5

    plt.subplot(2, 5, i+1)
    plt.imshow(image)
    plt.title(f"Actual: {actual_label}\nPredicted: {pred_label}")
    plt.axis('off')
plt.show()
```