Aim: The code aims to train a deep learning model, specifically a transfer learning approach using(ResNet in this scenario) , to classify images from Oxford 102 flowers dataset. By leveraging PyTorch and transfer learning, the goal is to create an effective classifier capable of distinguishing various flower species based on image data.

Description:

1. **Setup**:

   The code is designed to run in a Google Colab environment. It begins by configuring the Colab instance for Kaggle API use, allowing seamless dataset downloads directly from Kaggle.

2. **Data Preparation**:

   The code downloads dataset from Kaggle, then unzips and organizes it for model training and validation.

   Data transformations are applied to the training and validation datasets, including resizing, random horizontal flipping, and normalization to make the model training process more robust.

3. **Model Selection**:

   Using a pretrained ResNet model from PyTorch's model library and adapt it for flower classification through transfer learning. This process involves modifying the fully connected layers of the ResNet model to match the number of classes in the flower dataset.

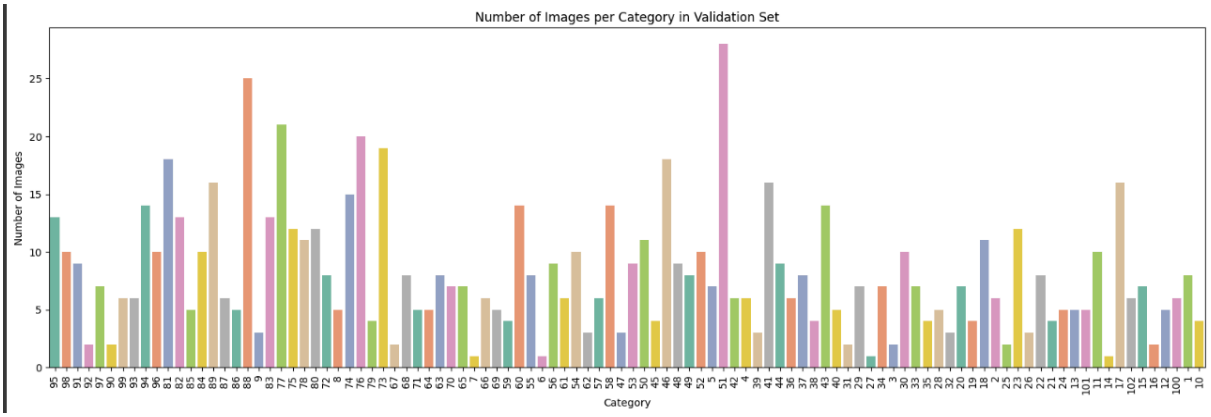
4. **Training**:

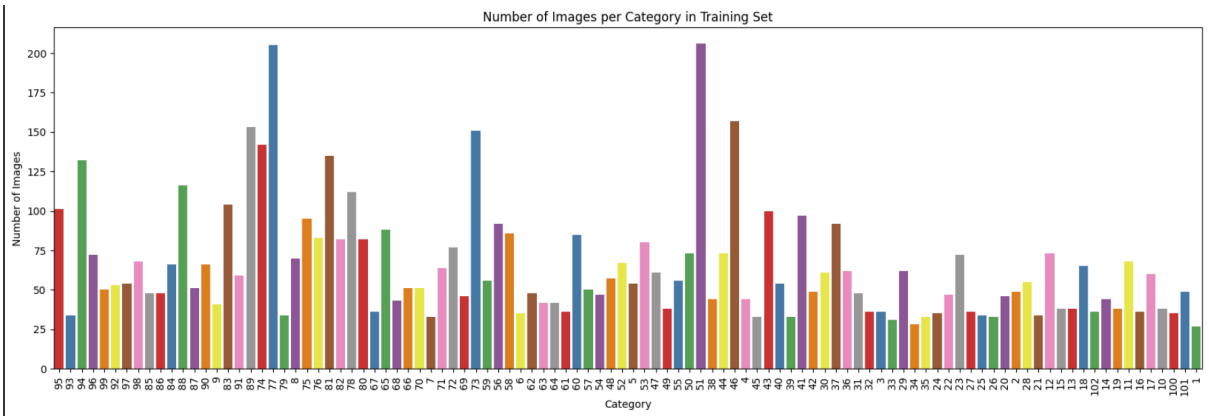
   With a specified optimizer and loss function, the code iteratively trains the model on the flower dataset, adjusting weights to minimize classification errors.

5. **Evaluation**:

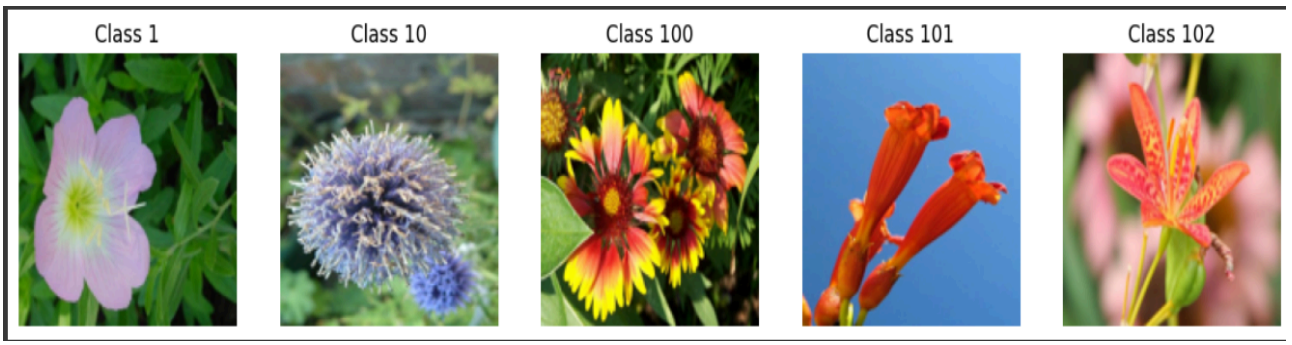
   After training, the model is validated on a separate portion of the dataset to assess its accuracy in classifying flower species.

Data Distribution in given Oxford dataset:





Visualisation of dataset sample images:



Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader, Dataset
import os


# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Define data transforms for training and validation
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}
```

```
# Custom Dataset to filter only 100 classes
class FilteredDataset(Dataset):
    def __init__(self, dataset):
        self.dataset = dataset
        self.filtered_indices = [i for i, (_, label) in enumerate(dataset) if label < 100]

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

    def __getitem__(self, idx):
        actual_idx = self.filtered_indices[idx]
        img, label = self.dataset[actual_idx]
        return img, label

# Load data
data_dir = '/content/dataset/dataset'
train_dataset = FilteredDataset(datasets.ImageFolder(os.path.join(data_dir, 'train'), data_transforms['train']))
val_dataset = FilteredDataset(datasets.ImageFolder(os.path.join(data_dir, 'valid'), data_transforms['val']))
dataloaders = {
    'train': DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=4),
    'val': DataLoader(val_dataset, batch_size=32, shuffle=False, num_workers=4)
}

# Function to display a batch of images with labels
def show_sample_images(dataloader, class_names):
```

```
    # Get a batch of images and labels
    images, labels = next(iter(dataloader))

    # Make a grid from batch and unnormalize images
    images = images[:8]  # Display 8 images for clarity
    out = torchvision.utils.make_grid(images, nrow=4)  # Arrange in a 2x4 grid
    out = out / 2 + 0.5  # Unnormalize

    plt.figure(figsize=(10, 5))
    plt.imshow(out.permute(1, 2, 0))  # Convert from Tensor image
    plt.axis('off')

    # Print labels
    labels = labels[:8]  # Only first 8 labels
    label_names = [class_names[label] for label in labels]
    print("Labels:", label_names)
# Load class names from the dataset
class_names = train_dataset.dataset.classes  # Assumes the classes are in the original dataset
# Show sample images from the training set
show_sample_images(dataloaders['train'], class_names)
# Show sample images from the training dataset
print("Sample images from the training dataset:")
# Load a pre-trained model (ResNet18)
model = models.resnet18(pretrained=True)
num_ftrs = model.fc.in_features
```

```
model.fc = nn.Linear(num_ftrs, 100)  # Adjusted to 100 classes

model = model.to(device)

# Set up loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

# Training function

def train_model(model, dataloaders, criterion, optimizer, num_epochs=5):

    for epoch in range(num_epochs):

        print(f'Epoch {epoch+1}/{num_epochs}')

        for phase in ['train', 'val']:

            model.train() if phase == 'train' else model.eval()

            running_loss = 0.0

            running_corrects = 0

            for inputs, labels in dataloaders[phase]:

                inputs, labels = inputs.to(device), labels.to(device)

                optimizer.zero_grad()



                with torch.set_grad_enabled(phase == 'train'):

                    outputs = model(inputs)

                    _, preds = torch.max(outputs, 1)

                    loss = criterion(outputs, labels)

                    if phase == 'train':

                        loss.backward()

                        optimizer.step()

                running_loss += loss.item() * inputs.size(0)
```

```
            running_corrects += torch.sum(preds == labels.data)
        epoch_loss = running_loss / len(dataloaders[phase].dataset)
        epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)
        print(f'{phase.capitalize()} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')
    return model
# Train the model
model_trained = train_model(model, dataloaders, criterion, optimizer, num_epochs=5)
# Evaluate the model and visualize predictions on the validation set
def visualize_predictions(model, dataloader, num_images=6):
    model.eval()
    images, labels, predictions = [], [], []
    with torch.no_grad():
        for i, (inputs, labels_batch) in enumerate(dataloader):
            inputs = inputs.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            images.append(inputs.cpu())
            labels.extend(labels_batch.cpu().numpy())
            predictions.extend(preds.cpu().numpy())
            if len(images) >= num_images:
                break
    for i in range(num_images):
        print(f'True: {labels[i]}, Pred: {predictions[i]}')
print("Sample predictions on the validation dataset:")
visualize_predictions(model_trained, dataloaders['val'], num_images=6)
```

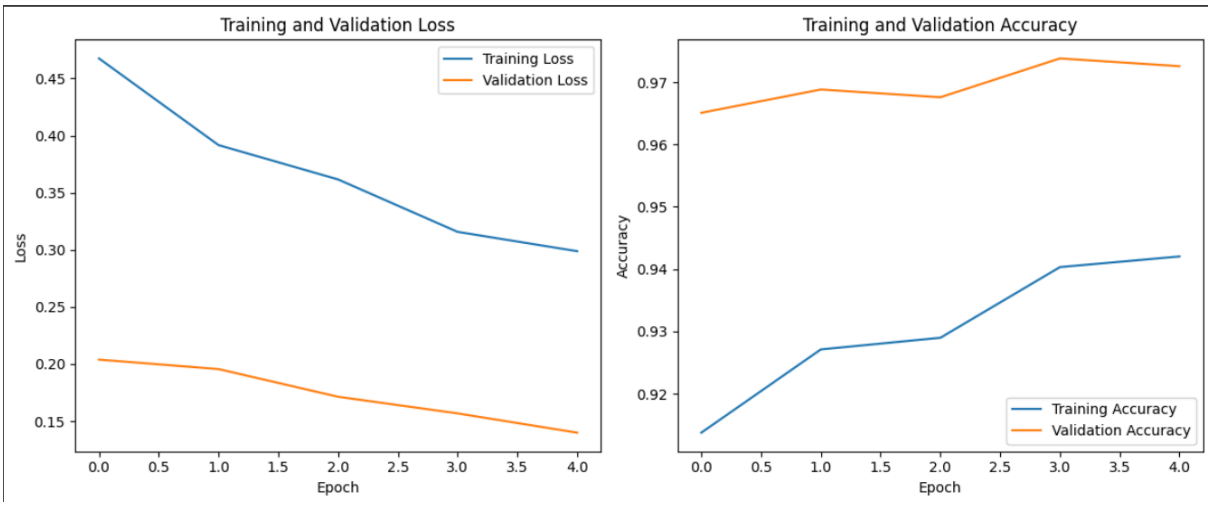
```
import matplotlib.pyplot as plt

def train_model(model, dataloaders, criterion, optimizer, num_epochs=5):
    train_losses, val_losses = [], []
    train_accuracies, val_accuracies = [], []
   for epoch in range(num_epochs):
        print(f'Epoch {epoch+1}/{num_epochs}')
        for phase in ['train', 'val']:
            model.train() if phase == 'train' else model.eval()
            running_loss = 0.0
            running_corrects = 0
            for inputs, labels in dataloaders[phase]:
                inputs, labels = inputs.to(device), labels.to(device)
                optimizer.zero_grad()
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            epoch_loss = running_loss / len(dataloaders[phase].dataset)
            epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)
            if phase == 'train':
```

```
            train_losses.append(epoch_loss)
            train_accuracies.append(epoch_acc.item())
        else:
            val_losses.append(epoch_loss)
            val_accuracies.append(epoch_acc.item())
        print(f'{phase.capitalize()} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')
    return model, train_losses, val_losses, train_accuracies, val_accuracies
# Retrain the model and collect metrics
model_trained, train_losses, val_losses, train_accuracies, val_accuracies = train_model(model, dataloaders, criterion, optimizer, num_epochs=5)
# Plotting the losses
plt.figure(figsize=(12, 5))
# Plot for Loss
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
# Plot for Accuracy
plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.xlabel('Epoch')
```

```
plt.ylabel('Accuracy')

plt.title('Training and Validation Accuracy')

plt.legend()

plt.tight_layout()

plt.show()
```

Outputs:

```
Epoch 1/5
/usr/local/lib/python3.10/dist-packages/torch/
  warnings.warn(
Train Loss: 0.4676 Acc: 0.9137
Val Loss: 0.2039 Acc: 0.9651
Epoch 2/5
Train Loss: 0.3917 Acc: 0.9271
Val Loss: 0.1956 Acc: 0.9688
Epoch 3/5
Train Loss: 0.3617 Acc: 0.9290
Val Loss: 0.1714 Acc: 0.9676
Epoch 4/5
Train Loss: 0.3157 Acc: 0.9403
Val Loss: 0.1568 Acc: 0.9738
Epoch 5/5
Train Loss: 0.2988 Acc: 0.9420
Val Loss: 0.1399 Acc: 0.9726
```



From this graphs we can observe training and validation loss differ greatly and model performance could be improved if we increase number of epochs.

```
Sample predictions on the validation dataset:
True: 0, Pred: 0
True: 0, Pred: 0
True: 0, Pred: 0
True: 0, Pred: 87
True: 0, Pred: 0
True: 0, Pred: 0
```

Conclusion

This code successfully demonstrates a robust workflow for building a deep learning model tailored to image classification. Leveraging transfer learning with a pretrained ResNet model simplifies and accelerates the training process. By adapting a pre-trained ResNet, the code bypasses the need for extensive computational resources and vast datasets typically required to train deep models from scratch.

Device Configuration: The code dynamically selects between GPU and CPU using PyTorch's device management feature, allowing compatibility across hardware environments. This device management is vital for scaling the code across different setups but adds complexity in ensuring consistency across devices.

Model Customization via Transfer Learning: Adapting the ResNet model involves replacing the final layer to match the number of flower classes. This transfer learning approach reduces training time, as only the final few layers need updating. However, balancing between freezing earlier layers and training later ones requires careful consideration, as training too many layers could increase computational costs without proportional performance gains.

Challenge: Ensuring the modified architecture maintains compatibility with the input image size and the specific dataset is key. Adjustments to layer structures require understanding of the model's internals and potential changes to training hyperparameters, which can complicate the training process.

UseCases: Knowledge gained in this case study can be used in some other problems in different domains. Hence this is called a transfer learning process.